Toward Millions of File System IOPS on Low-Cost, Commodity Hardware

Da Zheng, Randal Burns
Department of Computer Science
Johns Hopkins University

Alexander S. Szalay
Department of Physics and Astronomy
Johns Hopkins University

ABSTRACT

We describe a storage system that removes I/O bottlenecks to achieve more than one million IOPS based on a user-space file abstraction for arrays of commodity SSDs. The file abstraction refactors I/O scheduling and placement for extreme parallelism and non-uniform memory and I/O. The system includes a set-associative, parallel page cache in the user space. We redesign page caching to eliminate CPU overhead and lock-contention in non-uniform memory architecture machines. We evaluate our design on a 32 core NUMA machine with four, eight-core processors. Experiments show that our design delivers 1.23 million 512-byte read IOPS. The page cache realizes the scalable IOPS of Linux asynchronous I/O (AIO) and increases user-perceived I/O performance linearly with cache hit rates. The parallel, set-associative cache matches the cache hit rates of the global Linux page cache under real workloads.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Secondary storage

General Terms

Design, Performance

Keywords

Data-intensive computing, page cache optimization, millions of IOPS, low cost, solid-state storage devices

1. INTRODUCTION

Systems that perform fast, random I/O are revolutionizing commercial data services and scientific computing, creating the capability to quickly extract information from massive data sets [15]. For example, NoSQL systems underlying cloud stores generate small, incoherent I/Os that search key indexes and reference values, tables, documents, or graphs. The design of Amazon's DynamoDB testifies to this trend; it differentiates itself as a fast and scalable technology based on integrating SSDs into a key/value database [1]. In scientific computing, SSDs improve the throughput of graph and network analyses by an order of magnitude over magnetic disk [28]. Data sets that describe graphs are notoriously difficult to analyze on the steep memory hierarchies of conventional HPC hardware [14], because they induce fine-grained, incoherent data accesses. The future of data-driven computing will rely on extending random access to large-scale storage, building on today's SSDs and other non-volatile memories as they emerge.

Specialized hardware for random access offers an effective solution, albeit costly. For example, Fusion-IO provides NAND-flash persistent memory that delivers over one million accesses per second. Fusion-IO represents a class of persistent memory devices that are used as application accelerators integrated as memory addressed directly from the processor. As another approach, the Cray XMT architecture implements a flat memory system so that all cores have fast access to all memory addresses. This approach is limited by memory size. All custom hardware approaches cost multiples of commodity SSDs.

While recent advances in commodity SSDs have produced machines with hardware capable of over one million random IOPS, standard system configurations fail to realize the full potential of the hardware. Performance issues are ubiquitous in hardware and software, ranging from the assignment of interrupts, to non-uniform memory bandwidth, to lock contention in device drivers and the operating system. Problems arise because I/O systems were not designed for the extreme parallelism of multicore processors and SSDs. The design of file systems, page caches, device drivers and I/O schedulers does not reflect the parallelism (tens to hundreds of contexts) of the threads that initiate I/O or the multi-channel devices that service I/O requests.

None of the I/O access methods in Linux kernel perform well on a high-speed SSD array. I/O requests go through many layers in the kernel before reaching a device [12]. This produces significant CPU consumption under high IOPS. Each layer in the block subsystem uses locks to protect its data structures during concurrent updates. Furthermore, SSDs require many parallel I/Os to achieve optimal performance, while synchronous I/O, such as buffered I/O and direct I/O, issues one I/O request per thread at a time. The many threads needed to load the I/O system produce lock contention and high CPU consumption. Asynchronous I/O (AIO), which issues multiple requests in a single thread, provides a better option for accessing SSDs. However, AIO does

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC13 November 17-21, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00
http://dx.doi.org/10.1145/2503210.2503225.

not integrate with the operating system page cache so that SSD throughput limits user-perceived performance.

The goal of our system design is twofold: (1) to eliminate bottlenecks in parallel I/O to realize the full potential of SSD arrays and (2) to integrate caching into SSD I/O to amplify the user-perceived performance to memory rates. Although the performance of SSDs has advanced in the past years, it does not approach memory both in random IOPS or latency (Table 1). Furthermore, RAM may be accessed at a finer granularity 64 versus 512 bytes, which can widen the performance gap by another factor of eight for workloads that perform small requests. We conclude that SSDs require a memory page cache interposed between an SSD file system and applications. This is in contrast to translating SSD storage into the memory address space using direct I/O. A major obstacle to overcome is that the page caches in operating systems do not scale to millions of IOPS. They were designed for magnetic disks that perform only about 100 IOPS per device. Performance suffers as access rates increase owing to lock contention and with increased mutlicore parallelism owing to processor overhead.

The first contribution of this paper is the design of a user-space file abstraction that performs more than one million IOPS on commodity hardware. We implement a thin software layer that gives application programmers synchronous and asynchronous interfaces to file I/O. The system modifies I/O scheduling, interrupt handling, and data placement to reduce processor overhead, eliminate lock contention, and account for affinities between processors, memory, and storage devices.

We further present a scalable user-space cache for NUMA machines and arrays of SSDs that realizes I/O performance of Linux asynchronous I/O for cache misses and preserve the cache hit rates of the Linux page cache under real workloads. Our cache design is set-associative; it breaks the page buffer pool into a large number of small page sets and manages each set independently to reduce lock contention. The cache design extends to NUMA architectures by partitioning the cache by processors and using message passing for inter-processor communication.

The evaluation on a 32 core NUMA machine shows promising results. Our design delivers 1.23 million 512-byte read IOPS with our current hardware. The page cache realizes the scalable IOPS of Linux asynchronous I/O (AIO) and increases user-perceived I/O performance linearly with cache hit rates. To demonstrate the performance of our system in a typical HPC environment, we integrate it into the IOR benchmark [30], a popolar HPC benchmark for I/O performance. The IOR benchmark further shows our design achieves good performance for all request sizes and the synchronous write in our system performs nearly as well as asynchronous write.

2. RELATED WORK

This research falls into the broad area of the scalability operating systems with parallelism. Several research efforts [3, 32] treat a multicore machine as a network of independent cores and implement OS functions as a distributed system of processes that communicate with message passing. We embrace this idea for processors and hybridize it with traditional SMP programming models for cores. Specifically, we use shared memory for communication inside a processor and message passing between processors.

	random IOPS	latency	granularity
ioDrive Octal [13]	1,300,000	45μs	512B
OCZ Vertex 4 [23]	120,000	20μs	512B
DDR3-1333	7,300,000	15ns	64B

Table 1: The performance of specialized memory-addressable NAND flash (ioDrive Octal), a commodity SSD (OCZ Vertex 4), and memory (DDR3-1333). IOPS are measured with 512-byte random accesses.

As a counterpoint, a team from MIT [8] conducted a comprehensive survey on the kernel scalability and concluded that the traditional monolithic kernel can also have good parallel performance. We demonstrate that this is not the case for the page cache at millions of IOPS.

More specifically, our work relates to the scalable page caching. Yui et al. [33] designed a lock-free cache management for database based on Generalized CLOCK [31] and use a lock-free hashtable as index. They evaluated their design in a eight-core computer. We provide an alternative design of scalable cache and evaluate our solution at a larger scale.

The open-source community has improved the scalability of Linux page cache. Read-copy-update (RCU) [20] reduces contention through lock-free synchronization of parallel reads from the page cache (cache hits). However, the Linux kernel still relies on spin locks to protect page cache from concurrent updates (cache misses). In contrast, our design focuses on random I/O, which implies a high churn rate of pages into and out of the cache.

Park et al. [24] evaluated the performance effects of SSDs on scientific I/O workloads and they used workloads with large I/O requests. They concluded that SSDs can only provide modest performance gains over mechanical hard drives. As the advance of SSD technology, the performance of SSDs have been improved significantly, we demonstrate that our SSD array can provide random and sequential I/O performance many times faster than mechanical hard drives to accelerate scientific applications.

The set-associative cache was originally inspired by theoretical results that shows that a cache with restricted associativity can approximate LRU [29]. We build on this result to create a set-associative cache that matches the hit rates of the Linux kernel in practice.

The high IOPS of SSDs have revealed many performance issues with traditional I/O scheduling, which has lead to the development of new fair queuing techniques that work well with SSDs [25]. We also have to modify I/O scheduling as one of many optimizations to storage performance.

Our previous work [34] shows that a fixed size set-associative cache achieves good scalability with parallelism using a RAM disk. This paper extend this result to SSD arrays and adds features, such as replacement, write optimizations, and dynamic sizing. The design of the user-space file abstraction is novel to this paper as well.

3. A HIGH IOPS FILE ABSTRACTION

Although one can attach many SSDs to a machine, it is a non-trivial task to aggregate the performance of all SSDs. The default Linux configuration delivers only a fraction of optimal performance owing to skewed interrupt distribution,

device affinity in the NUMA architecture, poor I/O scheduling, and lock contention in Linux file systems and device drivers. The process of optimizing the storage system to realize the full hardware potential includes setting configuration parameters, the creation and placement of dedicated threads that perform I/O, and data placement across SSDs. Our experimental results demonstrate that our design improves system IOPS by a factor of 3.5.

3.1 Reducing Lock Contention

Parallel access to file systems exhibits high lock contention. Ext3/ext4 holds an exclusive lock on an inode, a data structure representing a file system object in the Linux kernel, for both reads and writes. For writes, XFS holds an exclusive lock on each inode that deschedules a thread if the lock is not immediately available. In both cases, high lock contention causes significant CPU overhead or, in the case of XFS, frequent context switch, and prevents the file systems from issuing sufficient parallel I/O. Lock contention is not limited to the file system, the kernel has shared and exclusive locks for each block device (SSD).

To eliminate lock contention, we create a dedicated thread for each SSD to serve I/O requests and use asynchronous I/O (AIO) to issue parallel requests to an SSD. Each file in our system consists of multiple individual files, one file per SSD, a design similar to PLFS [4]. By dedicating an I/O thread per SSD, the thread owns the file and the per-device lock exclusively at all time. There is no lock contention in the file system and block devices. AIO allows the single thread to output multiple I/Os at the same time. The communication between application threads and I/O threads is similar to message passing. An application thread sends requests to an I/O thread by adding them to a rendezvous queue. The add operation may block the application thread if the queue is full. Thus, the I/O thread attempts to dispatch requests immediately upon arrival. Although there is locking in the rendezvous queue, the locking overhead is reduced by the two facts: each SSD maintains its own message queue, which reduces lock contention; the current implementation bundles multiple requests in a single message, which reduces the number of cache invalidations caused by locking.

3.2 Processor Affinity

Non-uniform performance to memory and the PCI bus throttles IOPS owing to the inefficiency of remote accesses. In recent multi-processor machines for both AMD and Intel architectures, each processor connects to its own memory and PCI bus. The memory and PCI bus of remote processors are directly addressable, but at increased latency and reduced throughput.

We avoid remote accesses by binding I/O threads to the processors connected to the SSDs that they access. This optimization leverages our design of using dedicated I/O threads, making it possible to localize all requests, regardless of how many threads perform I/O. By binding threads to processors, we ensure that all I/Os are sent to the local PCI bus.

3.3 Other Optimizations

Distributing Interrupts: With the default Linux setting, interrupts from SSDs are not evenly distributed among processor cores and we often witness that all interrupts are sent to a single core. Such large a number of interrupts saturates a CPU core which throttles system-wide IOPS.

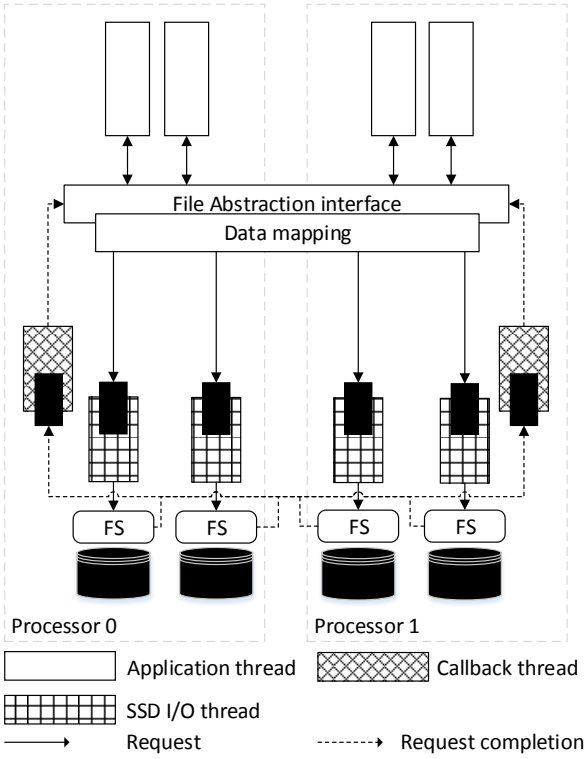


Figure 1: The architecture of the SSD file abstraction (SSDFA).

We remove this bottleneck by distributing interrupts evenly among all physical cores of a processor using the message signalled interrupts extension to PCI 3.0 (MSI-X) [21]. MSI-X allows devices to select targets for up to 2048 interrupts. We distribute the interrupts of a storage controller host-bus adapter across multiple cores of its local processor.

I/O scheduler: Completely Fair Queuing (CFQ), the default I/O scheduler in the Linux kernel >2.6.18, maintains I/O requests in per-thread queues and allocates time slices for each process to access disks to achieve fairness. When many threads access many SSDs simultaneously, CFQ prevent threads from delivering sufficient parallel requests to keep SSDs busy. Performance issues with CFQ and SSDs have lead researchers to redesign I/O scheduling [25]. Future Linux releases plan to include new schedulers.

At present, there are two solutions. The most common is to use the noop I/O scheduler, which does not perform per-thread request management. This also reduces CPU overhead. Alternatively, accessing an SSD from a single thread allows CFQ to inject sufficient requests. Both solutions alleviate the bottleneck in our system.

Data Layout: To realize peak aggregate IOPS, we parallelize I/O among all SSDs by distributing data. We offer three data distribution functions implemented in the data mapping layer of Figure 1.

- Striping: Data are divided into fixed-size small blocks placed on successive disks in increasing order. This layout is most efficient for sequential I/O, but susceptible to hotspots.
- Rotated Striping: Data are divided into stripes but

the start disk for each stripe is rotated, much like distributed parity in RAID5 [27]. This pattern prevents strided access patterns from skewing the workload to a single SSD.

- Hash mapping: The placement of each block is randomized among all disks. This fully declusters hotspots, but requires each block to be translate by a hash function.

Workloads that do not perform sequential I/O benefit from randomization.

3.4 Implementation

We implement this system in a user-space library that exposes a simple file abstraction (SSDFA) to user applications. It supports basic operations such as file creation, deletion, open, close, read and write, and provides both synchronous and asynchronous read and write interface. Each virtual file has metadata to keep track of the corresponding files on the underlying file system. Currently, it does not support directories.

The architecture of the system is shown in Figure 1. It builds on top of a Linux native file system on each SSD. Ext3/ext4 performs well in the system as does XFS, which we use in experiments. Each SSD has a dedicated I/O thread to process application requests. On completion of an I/O request, a notification is sent to a dedicated callback thread for processing the completed requests. The callback threads help to offload computation in the I/O threads and help applications to achieve processor affinity. Each processor has a callback thread.

4. A SET-ASSOCIATIVE PAGE CACHE

The emergence of SSDs has introduced a new performance bottleneck into page caching: managing the high churn or page turnover associated with the large number of IOPS supported by these devices. Previous efforts to parallelize the Linux page cache focused on parallel read throughput from pages already in the cache. For example, read-copy-update (RCU) [20] provides low-overhead wait free reads from multiple threads. This supports high-throughput to in-memory pages, but does not help address high page turnover.

Cache management overheads associated with adding and evicting pages in the cache limit the number of IOPS that Linux can perform. The problem lies not just in lock contention, but delays from the L1-L3 cache misses during page translation and locking. We redesign the page cache to eliminate lock and memory contention among parallel threads by using set-associativity. The page cache consists of many small sets of pages (Figure 2). A hash function maps each logical page to a set in which it can occupy any physical page frame.

We manage each set of pages independently using a single lock and no lists. For each page set, we retain a small amount of metadata to describe the page locations. We also keep one byte of frequency information per page. We keep the metadata of a page set in one or few cache lines to minimize CPU cache misses. If a set is not full, a new page is added to the first unoccupied position. Otherwise, a user-specified page eviction policy is invoked to evict a page. The current available eviction policies are LRU, LFU, Clock [11] and GClock [31].

As shown in figure 2, each page contains a pointer to a linked list of I/O requests. When a request requires a page

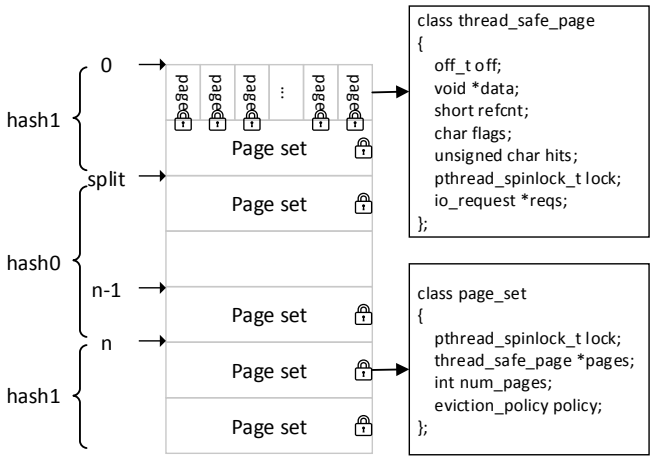


Figure 2: The organization of the set-associative cache showing the data structures and locks for pages and page sets. The hash0 and hash1 functions implement linear hashing [18] used to resize the cache. $n = init_size \times 2^{level}$.

for which an I/O is already pending, the request will be added to the queue of the page. Once I/O on the page is complete, all requests in the queue will be served.

There are two levels of locking to protect the data structure of the cache:

- per-page lock: a spin lock to protect the state of a page.
- per-set lock: a spin lock to protect search, eviction, and replacement inside a page set.

A page also contains a reference count that prevents a page from being evicted while the page is being used by other threads.

4.1 Resizing

A page cache must support dynamic resizing to share physical memory with processes and swap. We implement dynamic resizing of the cache with linear hashing [18]. Linear hashing proceeds in rounds that double or halve the hashing address space. The actual memory usage can grow and shrink incrementally. We hold the total number of allocated pages through loading and eviction within the page sets. When splitting a page set i, we rehash its pages to set i and $init_size \times 2^{level} + i$. The number of page sets is defined as $init_size \times 2^{level} + split$. *level* indicates the number of times that pages have been split. *split* points to the page set to be split. The cache uses two hash functions within each level *hash0* and *hash1*:

- $hash0(v) = h(v, init_size \times 2^{level})$
- $hash1(v) = h(v, init_size \times 2^{level+1})$

If the result of *hash0* is smaller than *split*, *hash1* is used for the page lookup as shown in figure 2.

4.2 Read and write optimizations

Even though SSDs deliver high random IOPS, they still have higher throughput for larger I/O requests [6]. Furthermore, accessing a block of data on an SSD goes through a long code path in the kernel and consumes a significant

number of CPU cycles [12]. By initiating larger requests, we can reduce CPU consumption and increase throughput.

Our page cache converts large read requests into a multi-buffer requests in which each buffer is single page in the page cache. Because we use the multi-buffer API of libaio, the pages need not be contiguous in memory. A large application request may be broken into multiple requests if some pages in the range read by the request are already in the cache or the request crosses a stripe boundary. The split requests are reassembled once all I/O completes and then delivered to the application as a single request.

The page cache has a dedicated thread to flush dirty pages. It selects dirty pages from the page sets where the number of dirty pages exceeds a threshold and write them with parallel asynchronous I/O to SSDs. Flushing dirty pages can reduce average write latency, which dramatically improves the performance of synchronous write issued by applications. However, the scheme may also increase the amount of data written to SSDs. To reduce the number of dirty pages to be flushed, the current policy within a page set is to select the dirty pages that are most likely to be evicted in a near future.

To reduce write I/O, we greedily flush all adjacent dirty pages using a single I/O, including pages that have not yet been scheduled for writeback. This optimization was originally proposed in disk file systems [2]. The hazard is that flushing pages early will generate more write I/O when pages are being actively written. To avoid generating more I/O, we tweak the page eviction policy, similar to CFLRU [26], to keep dirty pages in the memory longer: when the cache evicts a page from a set, it tries to evict a clean page if possible.

4.3 NUMA design

Performance issues arise when operating a global, shared page cache on a non-uniform memory architecture. The problems stem from the increased latency of remote memory access, the reduced throughput of remote bulk memory copy [17]. A global, shared page cache treats all devices and memory uniformly. In doing so, it creates increasingly many remote operations as we scale the number of processors.

We extend the set-associative cache for the NUMA architectures (NUMA-SA) to optimize for workloads with relatively high cache hit rates and tackle hardware heterogeneity. The NUMA-SA cache design was inspired by multi-core operating systems that treat each core a node in a message-passing distributed system [3]. However, we hybridize this concept with standard SMP programming models: we use message passing for inter-processor operations but use shared-memory among the cores within each processor. Figure 3 shows the design of NUMA-SA cache. Each processor attached to SSDs has threads dedicated to performing I/O for each SSD. The dedicated I/O thread removes contention for kernel and file locks. The processors without SSDs maintain page caches to serve applications I/O requests.

I/O requests from applications are routed to the caching nodes through message passing to reduce remote memory access. The caching nodes maintain message passing queues and a pool of threads for processing messages. On completion of an I/O request, the data is written back to the destination memory directly and then a reply is sent to the issuing thread. This design opens opportunities to move application computation to the cache to reduce remote memory access.

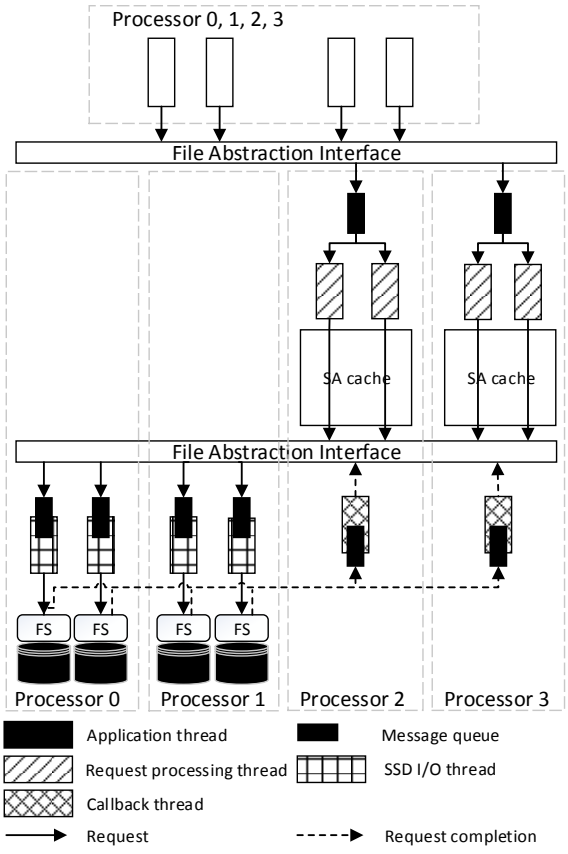


Figure 3: The architecture of the NUMA-SA cache on a four processor machine with two processors attached to SSDs. The page cache accesses SSDs with the same file abstraction interface as the one used by applications and expose the file abstraction interface to user applications.

We separate I/O nodes from caching nodes in order to balance computation. I/O operations require significant CPU and running a cache on an I/O node overloads the processor and reduces IOPS. This is a design decision, not a requirement, i.e. we can run a set-associative cache on the I/O nodes as well. In a NUMA machine, a large fraction of I/Os require remote memory transfers. This happens when application threads run on other nodes than I/O nodes. Separating the cache and I/O nodes does increase remote memory transfers. However, balanced CPU utilization makes up for this effect in performance. As systems scale to more processors, we expect that few processors will have PCI buses, which will increase the CPU load on these nodes, so that splitting these functions will continue to be advantageous.

Message passing creates many small requests and synchronizing these requests can become expensive. Message passing may block sending threads if their queue is full and receiving threads if their queue is empty. Synchronization of requests often involves cache line invalidation on shared data and thread rescheduling. Frequent thread rescheduling wastes CPU cycles, preventing application threads from getting enough CPU. We reduce synchronization overheads

Linux I/O scheduler	noop
Page cache size	512MB
Page eviction policy	GClock
Block size	16 pages
Block mapping	striping (SMP)/hash (NUMA)
Page set size	12 pages
AIO depth	32
Number of app threads	16
File system on SSDs	XFS

Table 2: Default configuration of experiments.

by amortizing them over larger messages.

5. EVALUATION

We evaluate our implementation with various workloads. We start with the evaluation on the user-space file abstraction without caching to demonstrate the raw performance of the hardware. We evaluate the page cache in the file abstraction with various workloads from commercial data services and scientific applications. We further integrate our system into the IOR benchmark [30] to measure its performance in a typical HPC environment.

We conduct experiments on a non-uniform memory architecture machine with four Intel Xeon E5-4620 processors, clocked at 2.2GHz, and 512GB memory of DDR3-1333. Each processor has eight cores with hyperthreading enabled, resulting in 16 logical cores. Only two processors in the machine have PCI buses connected to them. The machine has three LSI SAS 9217-8i host bus adapters (HBA) connected to a SuperMicro storage chassis, in which 16 OCZ Vertex 4 SSDs are installed. In addition to the LSI HBAs, there is one RAID controller that connects to disks with root filesystem. The machine runs Ubuntu Linux 12.04 and Linux kernel v3.2.30.

To compare the best performance of our system design with that of the Linux, we measure the system in two configurations: an SMP architecture using a single processor and NUMA using all processors. On all I/O measures, Linux performs best from a single processor. Remote memory operations make using all four processors slower.

- SMP configuration: 16 SSDs connect to one processor through two LSI HBAs controlling eight SSDs each. All threads run on the same processor. Data are striped across SSDs.
- NUMA configuration: 16 SSDs are connected to two processors. Processor 0 has five SSDs attached to an LSI HBA and one through the RAID controller. Processor 1 has two LSI HBAs with five SSDs each. Application threads are evenly distributed across all four processors. Data are distributed through a hash mapping that assigns 10% more I/Os to the LSI HBA attached SSDs. The RAID controller is slower.

Experiments use the configurations shown in Table 2 if not stated otherwise.

5.1 User-Space File Abstraction

This section enumerates the effectiveness of the hardware and software optimizations implemented in the SSD user-space file abstraction without caching, showing the contribution of each. The size of the smallest requests issued by the page cache is 4KB, so we focus on 4KB read and write

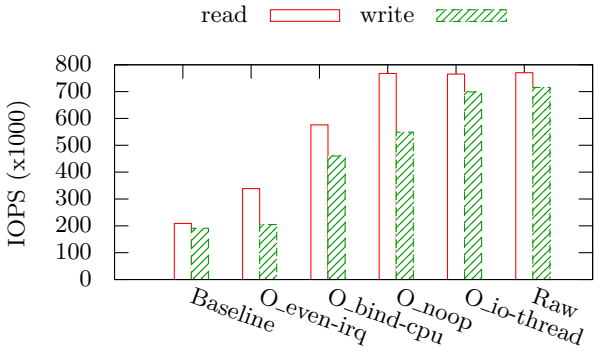


Figure 4: Optimizing page I/O on the SSD file abstraction accessed from an 8 core processor (SMP). The bars show the aggregate IOPS when applying four optimizations successively in comparison with the hardware's capabilities (raw).

performance. In each experiment, we read/write 40GB data randomly through the SSD file abstraction in 16 threads.

We perform four optimizations on the SSD file abstraction in succession to optimize performance.

- **O_even-irq:** distribute interrupts evenly among all CPU cores;
- **O_bind-cpu:** bind threads to the processor local to the SSD;
- **O_noop:** use the noop I/O scheduler;
- **O_io-thread:** create a dedicated I/O threads to access each SSD on behalf of the application threads.

Figure 4 shows I/O performance improvement of the SSD file abstraction when applying these optimizations in succession. Performance reaches a peak 765,000 read IOPS and 699,000 write IOPS from a single processor up from 209,000 and 191,000 IOPS unoptimized. Distributing interrupts removes a CPU bottleneck for read. Binding threads to the local processor has a profound impact, doubling both read and write by eliminating remote operations. Dedicated I/O threads improves write throughput, which we attribute to removing lock contention on the file system's inode.

When we apply all optimizations, the system realizes the performance of raw SSD hardware, as shown in Figure 4. It only loses less than 1% random read throughput and 2.4% random write throughput. The performance loss mainly comes from disparity among SSDs, because the system performs at the speed of the slowest SSD in the array. When writing data to SSDs, individual SSDs slow down due to garbage collection, which causes the entire SSD array to slow down. Therefore, write performance loss is higher than read performance loss. These performance losses compare well with the 10% performance loss measured by Caulfield [9].

When we apply all optimizations in the NUMA configuration, we approach the full potential of the hardware, reaching 1.23 million read IOPS. We show performance alongside the the Fusion-IO ioDrive Octal [13] for a comparison with state of the art memory-integrated NAND flash products (Table 3). This reveals that our design realizes comparable read performance using commodity hardware. SSDs have a 4KB minimum block size so that 512 bytes write a partial block and, thus, slow. The 766K 4KB writes offer a better point of comparison.

	SSDFA	ioDrive Octal 5TB
Read IOPS (512B)	1,228,100	1,190,000
Write IOPS (512B)	386,976	1,180,000
Read IOPS (4KB)	946,700	N/A
Write IOPS (4KB)	766,082	N/A
Read Bandwidth (64 kB)	6.8GB/s	6.0 GB/s
Write Bandwidth (64 kB)	5.6GB/s	4.4 GB/s

Table 3: The performance of NUMA SSD user-space file abstraction compared with FusionIO ioDrive Octal.

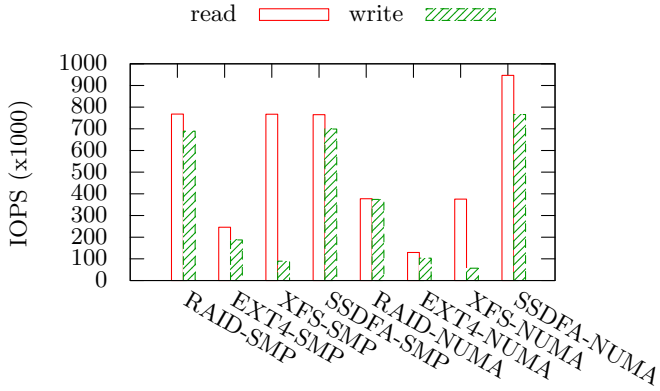


Figure 5: Performance of our user-space file abstraction with Linux file systems and software RAID. All systems use optimizations O_even-irq and O_noop.

We further compare our system with Linux software options, including block interfaces (software RAID) and file systems (Figure 5). Although software RAID can provide comparable performance in SMP configurations, NUMA results in a performance collapse to less than half the IOPS. Locking structures in file systems prevent scalable performance on Linux software RAID. Ext4 holds a lock to protect its data structure for both reads and writes. Although XFS realizes good read performance, it performs poorly for writes due to the exclusive locks that deschedule a thread if they are not immediately available.

As an aside, we see a performance decrease in each SSD as more SSDs are accessed in a HBA, as shown in Figure 6. A single SSD can deliver 73,000 4KB-read IOPS and 61,000 4KB-write IOPS, while eight SSDs in a HBA deliver only 47,000 read IOPS and 44,000 write IOPS per SSD. Other work confirms this phenomena [12]. The aggregate IOPS of an SSD array increases as the number of SSDs increases. Multiple HBAs scale. Performance degradation may be caused by lock contention in the HBA driver as well as by the interfere inside the hardware itself. As a design rule, we attach as few SSDs to a HBA as possible to increase the overall I/O throughput of the SSD array in the NUMA configuration.

5.2 Set-Associative Caching

We demonstrate the performance of set-associative and NUMA-SA caches under different workloads to illustrate their overhead and scalability and compare performance with the Linux page cache.

We choose workloads that exhibit high I/O rates and ran-

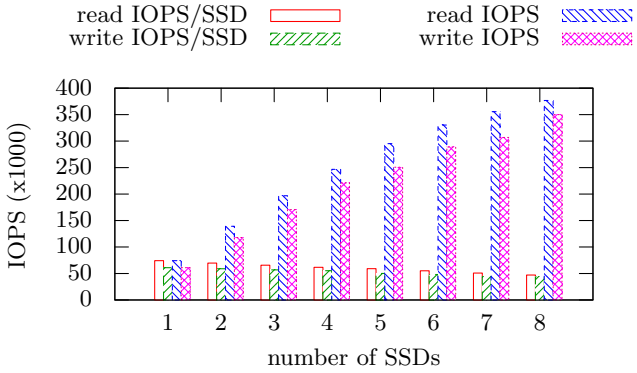


Figure 6: The 4KB read and write IOPS of individual SSDs and the aggregate IOPS of the SSD array with different numbers of SSDs in the array. All SSDs connect to a single HBA.

dom access that are representatives of cloud computing and data-intensive science. We generated traces by running applications, capturing I/O system calls, and converting them into file accesses in the underlying data distribution. System call traces ensure that I/O are not filtered by a cache. Workloads include:

- Uniformly random: The workload samples 128 bytes from pages chosen randomly without replacement. The workload generates no cache hits, accessing 10,485,760 unique pages with 10,485,760 physical reads.
- Yahoo! Cloud Serving Benchmark (YCSB) [10]: We derived a workload by inserting 30 million items into MemcacheDB and performing 30 million lookups according to YCSB's read-only Zipfian workload. The workload has 39,188,480 reads from 5,748,822 pages. The size of each request is 4096 bytes.
- Neo4j [22]: This workload injects a LiveJournal social network [19] in Neo4j and searches for the shortest path between two random nodes with Dijkstra algorithm. Neo4j sometimes scans multiple small objects on disks with separate reads, which biases the cache hit rate. We merge small sequential reads into a single read. With this change, the workload has 22,450,263 reads and 113 writes from 1,086,955 pages. The request size varies from 1 bytes to 1,001,616 bytes. Most requests are small. The mean request size is 57 bytes.
- Synapse labelling: This workload was traces at the Open Connectome Project `openconnecto.me` and describes the output of a parallel computer-vision pipeline run on a 4 Teravoxel image volume of mouse brain data. The pipeline detects 19 million synapses (neural connections) that it writes to spatial database. Write throughput limits performance. The workload labels 19,462,656 synapses in a 3-d array using 16 parallel threads. The workload has 19,462,656 unaligned writes of about 1000 bytes on average and updates 2,697,487 unique pages.

For experiments with multiple application threads, we dynamically dispatch small batches of I/O using a shared work queue so that all threads finish at nearly the same time, regardless of system and workload heterogeneity.

We measure the performance of Linux page cache with careful optimizations. We install Linux software RAID on

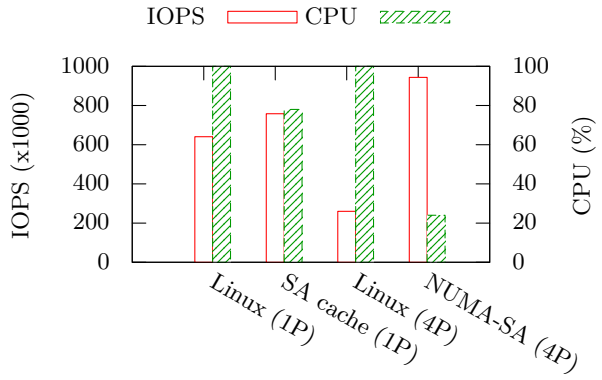


Figure 7: IOPS and CPU for random read (0% cache hit rate).

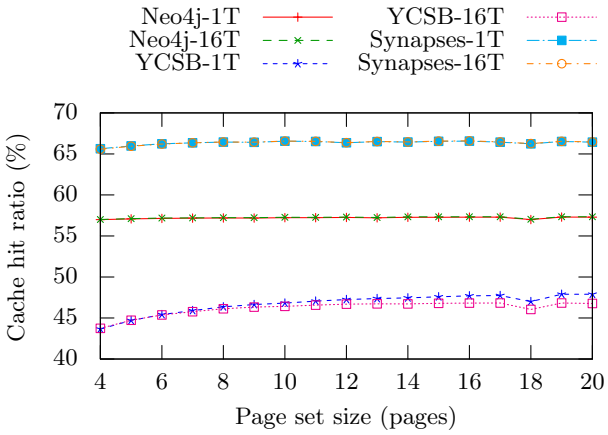


Figure 9: The impact of page set sizes on cache hit rate in the set-associative cache under real workloads both in one and 16 threads.

the SSD array and install XFS on software RAID. We run 256 threads to issue requests in parallel to Linux page cache in order to provide sufficient I/O requests to the SSD array. We disable read ahead to avoid the kernel to read unnecessary data. Each thread opens the data file by itself because concurrent updates on a file handler in a NUMA machine leads to expensive inter-processor cache line invalidation. As shown in the previous section, XFS does not support parallel write, we only measure read performance.

Random Workloads:

The first experiment demonstrates that set-associative caching relieves the processor bottleneck on page replacement. We run the uniform random workload with no cache hits and measure IOPS and CPU utilization (Figure 7). CPU cycles bound the IOPS of the Linux cache when run from a single processor—its best configuration. Linux uses all cycles on all 8 CPU cores to achieves 641K IOPS. The set-associative cache on the same hardware runs at under 80% CPU utilization and increases IOPS by 20% to the maximal performance of the SSD hardware. Running the same workload across the entire machine increases IOPS by another 20% to almost 950K for NUMA-SA. The same hardware configuration for Linux results in an IOPS collapse. Besides the poor performance of software RAID, a NUMA machine also amplifies locking overhead on the Linux page cache. The severe lock contention in the NUMA machine is caused by higher parallelism and more expensive cache line invalidation.

A comparison of IOPS as a function of cache hit rate reveals that the set-associative caches outperform the Linux cache at high hit rates and that caching is necessary to realize application performance. We measure IOPS under the uniform random workload for the Linux cache, with set-associative caching, and without caching (SSDFA). Overheads in the the Linux page cache make the set-associative cache realize roughly 30% more IOPS than Linux at all cache hit rates (Figure 8(a)). The overheads come from different sources at different hit rates. At 0% the main overhead comes from I/O and cache replacement. At 95% the main overhead comes from the Linux virtual file system [7] and page lookup on the cache index.

Non-uniform memory widens the performance gap (Figure 8). In this experiment application threads run on all processors. NUMA-SA effectively avoids lock contention and reduces remote memory access, but Linux page cache has severe lock contention in the NUMA machine. This results in a factor of four improvement in user-perceived IOPS when compared with the Linux cache. Notably, the Linux cache does not match the performance of our SSD file abstraction (with no cachcing) until a 75% cache hit rate, which reinforces the concept that lightweight I/O processing is equally important as caching to realize high IOPS.

The user-perceived I/O performance increases linearly with cache hit rates. This is true for set-associative caching, NUMA-SA, and Linux. The amount of CPU and effectiveness of the CPU dictates relative performance. Linux is always CPU bound.

The Impact of Page Set Size:

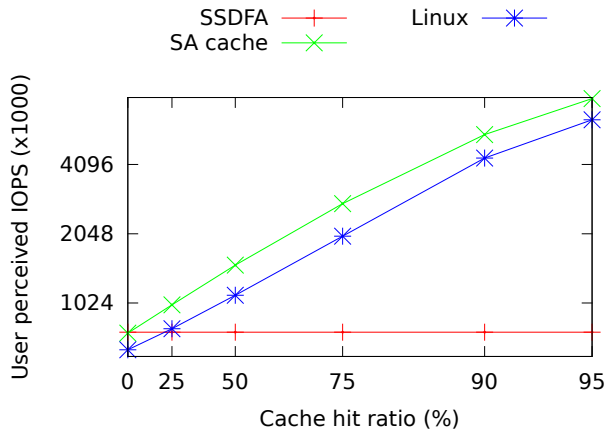
An important parameter in a set-associative cache is the size of a page set. The parameter defines a tradeoff between cache hit rate and CPU overhead within a page set. Smaller pages sets reduce cache hit rate and interference. Larger page sets better approximate global caches, but increase contention and the overhead of page lookup and eviction.

The cache hit rates provide a lower bound on the page set size. Figure 9 shows that the page set size has a limited impact on the cache hit rate. Although a larger page set size increases the hit rate in all workloads, it has more noticeable impact on the YCSB workload. Once the page set size increase beyond 12 pages per set, there are minimal benefits to cache hit rates.

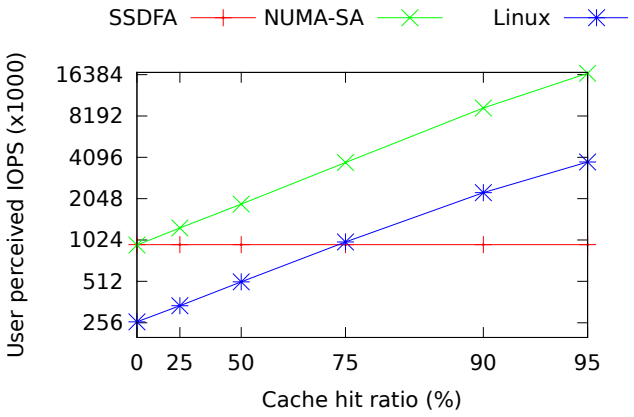
We choose the smallest page set size that provides good cache hit rates across all workloads. CPU overhead dictates small page sets. CPU increases with page set size by up to 4.3%. Cache hit rates result in better user-perceived performance by up to 3%. We choose 12 pages as the default configuration and use it for all subsequent experiments.

Cache Hit Rates:

We compare the cache hit rate of the set-associative cache with other page eviction policies in order to quantify how well a cache with restricted associativity emulates a global cache [29] on a variety of workloads. Figure 10 compares the Clock-Pro page eviction variant used by Linux [16]. We also include the cache hit rate of GClock [31] on a global page buffer. For the set-associative cache, we implement these replacement policies on each page set as well as least-



(a) SMP: one processor, eight applications cores.



(b) NUMA: four processors, 32 application cores.

Figure 8: User-perceived IOPS as a function of cache hit rate.

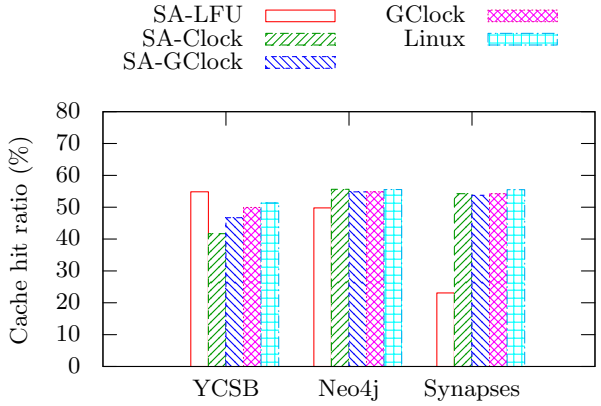


Figure 10: The cache hit rate of different cache designs under different workloads.

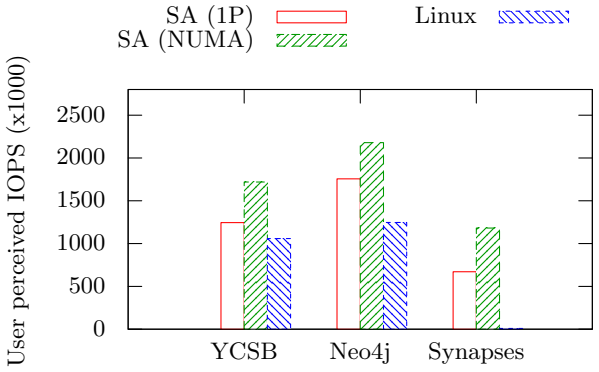


Figure 11: The performance of the set-associative cache on real workloads

frequently used (LFU). When evaluating the cache hit rate, we use the first half of a sequence of accesses to warm the cache and the second half to evaluate the hit rate.

The set-associative has a cache hit rate comparable to a global page buffer. It may lead to lower cache hit rate than a global page buffer for the same page eviction policy, as shown in the YCSB case. For workloads such as YCSB, which are dominated by frequency, LFU can generate more cache hits. It is difficult to implement LFU in a global page buffer, but it is simple in the set-associative cache due to the small size of a page set. We refer to [34] for more detailed description of LFU implementation in the set-associative cache.

Performance on Real Workloads:

For user-perceived performance, the increased IOPS from hardware overwhelms any losses from decreased cache hit rates. Figure 11 shows the performance of set-associative and NUMA-SA caches in comparison to Linux's best performance under the Neo4j, YCSB, and Synapse workloads, Again, the Linux page cache performs best on a single processor.

The set-associative cache performs much better than Linux page cache under real workloads. The Linux page cache achieves around 50–60% of the maximal performance for read-only workloads (Neo4j and YCSB). Furthermore, it delivers only 8,000 IOPS for an unaligned-write workload (Synapses). The poor performance of Linux page cache results from the exclusive locking in XFS, which only allows one thread to access the page cache and issue one request at a time to the block devices.

5.3 HPC benchmark

This section evaluates the overall performance of the user-space file abstraction under HPC benchmarks. The typical setup of some HPC benchmarks such as MADbench2 [5] has very large read/writes (in the order of magnitude of 100 MB). However, our system is optimized mainly for small random I/O accesses and requires many parallel I/O requests to achieve maximal performance. We choose the IOR benchmark [30] for its flexibility. IOR is a highly parameterized benchmark and Shan et al. [30] has demonstrated that IOR can reproduce diverse scientific workloads.

However, IOR has some limitations. It only supports multi-process parallelism and synchronous I/O interface. SSDs require many parallel I/O requests to achieve maximal performance, and the current implementation of our system can only share page cache among threads. To better assess

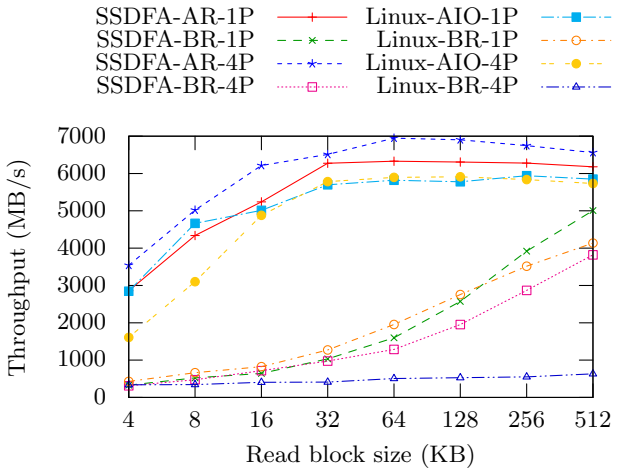


Figure 12: The read throughput of SSDFA and Linux solutions both on a single processor and on four processors, measured by the N-1 test of IOR. SSDFA-AR shows the throughput of asynchronous read (no cache), SSDFA-BR shows the throughput of (synchronous) buffered read. Linux-AIO and Linux-BR show the throughput of Linux AIO read and Linux buffered read.

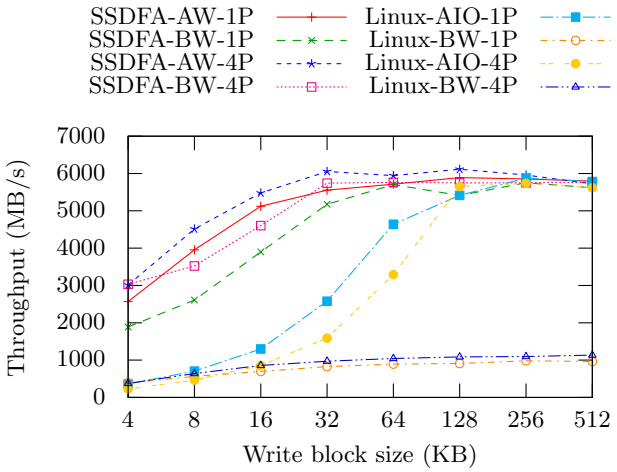


Figure 13: The write throughput of SSDFA and Linux solutions both on a single processor and on four processors, measured by the N-1 test of IOR. SSDFA-AW shows the throughput of asynchronous write (no cache), SSDFA-BW shows the throughput of SSDFA (synchronous) buffered write. Linux-AIO and Linux-BW shows the throughput of Linux AIO write and Linux buffered write.

the performance of our system, we add multi-threading and asynchronous I/O support to the IOR benchmark.

We perform thorough evaluations to our system with the IOR benchmark. We evaluate the synchronous and asynchronous interface of the SSD user-space file abstraction with various request sizes. We compare our system with Linux's existing solutions, software RAID and Linux page cache. For fair comparison, we only compare two options: asynchronous I/O without caching and synchronous I/O with caching, because Linux AIO does not support caching and our system currently does not support synchronous I/O without caching. We only evaluate SA cache in SSDFA because NUMA-SA cache is optimized for the asynchronous I/O interface and a high cache hit rate, and the IOR workload does not generate cache hits. We turn on the random option in the IOR benchmark. We use both the N-1 test (N clients read/write to a single file) and the N-N test (N clients read/write to N files) in IOR. Figure 12 and 13 show the results of the N-1 test. We only present the write performance in the N-N test shown in Figure 14 because the read performance in N-N test is similar to the one in N-1 test except that Linux buffered read in the N-N test can perform better in the NUMA configuration.

Figure 12 shows that SSDFA read significantly outperforms Linux read on a NUMA machine in the N-1 test. When the request size is small, Linux AIO read has much lower throughput than SSDFA asynchronous read (no cache) in the NUMA configuration due to the bottleneck in the Linux software RAID. Although SSDFA synchronous buffered read perform poorly with small request sizes, its throughput increases constantly with larger requests. The throughput of Linux buffered read in the NUMA configuration, on the other hand, barely increases with the request size due to the high cache overhead.

SSDFA write performs much better than all Linux's solutions in the N-1 test, especially for small request sizes (Figure 13). Thanks to our novel design of the flush thread in our page cache, SSDFA synchronous buffered write can achieve performance close to SSDFA asynchronous write. XFS has two exclusive locks on each file: one is to protect the inode data structure and is held briefly at each acquisition; the other is to protect I/O access to the file and is held for a longer time. Linux AIO write only acquires the one for inode and Linux buffered write acquires both locks. Thus, Linux AIO cannot perform well with small writes on XFS, but it can still reach maximal performance with a large request size on both a single processor and four processors. Linux buffered write, on the other hand, performs much worse and its performance can only be improved slightly with a larger request size.

Figure 14 shows that XFS and Linux page cache perform better in the N-N test than in the N-1 test, but SSDFA can still outperform them, especially for synchronous write. The N-N test essentially removes nearly all lock contention in XFS and Linux page cache, which boosts their performance. However, the bottleneck in the Linux software RAID still exists, so like the N-1 test, Linux AIO write is still slower than SSDFA asynchronous write in the NUMA configuration. Again, SSDFA synchronous buffered write can significantly outperform Linux buffered write thanks to our new design of the flush thread. Linux buffered write on a single processor performs worse than on four processors because Linux page cache hits a bottleneck in page allocation on a single processor. When a dedicated per-processor kernel thread cannot free sufficient pages to keep up the speed of page allocation required by other threads, all threads have to try to free pages and causes extremely high lock contention.

6. CONCLUSIONS

We present a storage system that achieves more than one

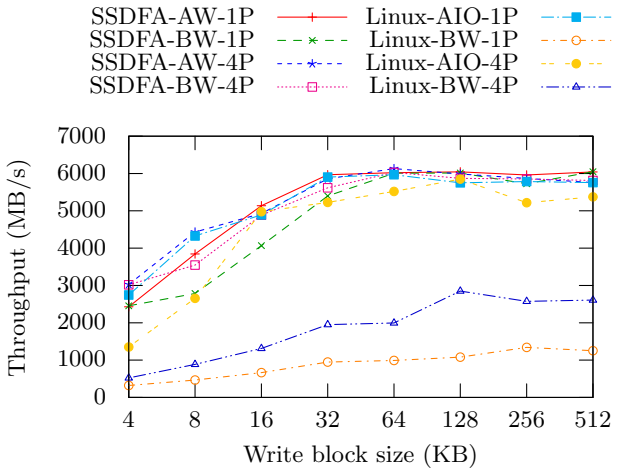


Figure 14: The write throughput of SSDFA and Linux solutions both on a single processor and on four processors, measured by the N-N test of IOR. SSDFA-AW shows the throughput of asynchronous write (no cache), SSDFA-BW shows the throughput of SSDFA (synchronous) buffered write. Linux-AIO and Linux-BW shows the throughput of Linux AIO write and Linux buffered write.

million random read IOPS based on a user-space file abstraction running on an array of commodity SSDs. The file abstraction builds on top of a local file system on each SSD in order to aggregates their IOPS. It also creates dedicated threads for I/O to each SSD. These threads access the SSD and file exclusively, which eliminates lock contention in the file and device interfaces. The design amplifies IOPS by 3.5 times and realizes nearly the full potential of the SSD hardware, less than 1% loss for reads and 2.4% for writes.

In the file abstraction, we deploy a set-associative parallel page cache designed for non-uniform memory architectures. The design divides the global page cache into many small, independent sets, which reduces lock contention. For NUMA architectures, the design minimizes the CPU overhead associated with remote memory copies through a hybrid SMP and message passing programming model. Each processor is treated as a node in a distributed system and inter-processor operations exchange messages through rendezvous queues served by a dedicated thread pool. The multiple cores of each processor are programmed as an SMP. With page caching, user-perceived throughput grows linearly with the cache hit rate up to 16 million IOPS, more than four times that realized by Linux. Our optimizations in the parallel page cache achieve good performance for all request sizes and synchronous write performs nearly as well as asynchronous write.

As a whole, the design alleviates bottlenecks associated with lock contention, CPU overhead, and remote memory copies across many layers of hardware and software. The design captures parallelism and non-uniform performance of modern hardware to realize world-class performance for commodity SSDs.

7. REFERENCES

[1] AmazonWebServices. Amazon dynamodb overview, a fully managed nosql database service. Available at `http://www.youtube.com/watch?v=oz-7wJJ9HZ0`, 2011.

[2] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey. AWOL: An adaptive write optimizations layer. In *Conference on File and Storage Technologies*, 2008.

[3] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. SchÃijpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Symposium on Operating Systems Principles*, 2009.

[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[5] J. Borrill, L. Oliker, J. Shalf, and H. Shan. Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2007.

[6] L. Bouganim, B. Jónssonn, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *Fourth Biennial Conference on Innovative Data Systems Research*, 2009.

[7] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[8] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Conference on Operating systems design and implementation*, 2010.

[9] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson. Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[10] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Symposium on Cloud computing*, 2010.

[11] F. J. Corbato. A paging experiment with the multics system. 1969.

[12] A. Foong, B. Veal, and F. Hady. Towards SSD-ready enterprise platforms. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2010.

[13] Fusion-IO ioDrive Octal. `http://www.fusionio.com/platforms/iodrive-octal/`, Accessed 3/11/2013.

[14] B. Hendrickson. Data analytics and high performance computing: When worlds collide. In *Los Alamos Computer Science Symposium*, 2009.

[15] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[16] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In

USENIX Annual Technical Conference, 2005.

[17] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman. NUMA-aware algorithms: the case of data shuffling. In *The biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[18] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Sixth International Conference on Very Large Data Bases*, 1980.

[19] Livejournal social network. `http://snap.stanford.edu/data/soc-LiveJournal1.html`, Accessed 3/11/2013.

[20] P. McKenney, D. Sarma, A. Arcangeli, A. Kleen, and O. Krieger. Read-copy update. In *Linux Symposium*, 2002.

[21] MSI-HOWTO. `http://lwn.net/Articles/44139/`, Accessed 3/6/2012.

[22] Neo4j. `www.neo4j.org`, Accessed 3/11/2013.

[23] OCZ VERTEX 4. `http://www.ocztechnology.com/ocz-vertex-4-sata-iii-2-5-ssd.html`, Accessed 3/11/2013.

[24] S. Park and K. Shen. A performance evaluation of scientific I/O workloads on flash-based SSDs. In *IEEE International Conference on Cluster Computing and Workshops.*, 2009.

[25] S. Park and K. Shen. FIOS: A fair and efficient I/O scheduler. In *Conference on File and Storage Technology*, 2012.

[26] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, New York, NY, USA, 2006.

[27] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD international conference on Management of data*, 1988.

[28] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Supercomputing*, 2010.

[29] S. Sen, S. Chatterjee, and N. Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6), 2002.

[30] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Austin, TX, 2008.

[31] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3), 1978.

[32] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. In *ACM SIGOPS Operating System Review (OSR)*, 2009.

[33] M. Yui, J. Miyazaki, S. Uemura, and H. Yamana. Nb-GCLOCK: A non-blocking buffer management based on the generalized CLOCK. In *International Conference on Data Engineering (ICDE)*, 2010.

[34] D. Zheng, R. Burns, and A. S. Szalay. A parallel page cache: IOPS and caching for multicore systems. In *USENIX conference on Hot Topics in Storage and File Systems*, 2012.